JINI Service Replication in Telecommunications

Niall Gallagher, Seamus O'Shea
University of Limerick, Limerick, Ireland

Abstract—**In this paper the emerging role of Java in Telecommunications is surveyed. In particular, attention is drawn to the advantages and disadvantages of the language for application and service development. The JINI framework for the construction of robust and self-configuring distributed systems is briefly described. Replication (of processes/data) is a desirable feature of any distributed system, whereby overloaded servers may be replicated elsewhere in the distributed system in order to maintain a given quality of service to clients. Several possible options for a JINI replication service are explored. A replication framework for JINI is described, and the contribution of such a replication service in a telecommunications environment is highlighted.**

I. Introduction

In its short history, Java has made a great impact in many application areas. Its platform independence, its downloadability, (e.g for protocol support) and its object oriented nature are very appealing characteristics. Platform independence means that software can be written once, and through the use of a Virtual Machine (VM) can be run on any target system. Already, Java has made an impact in end-user application software, in call control applications, in Intelligent Network (IN) services, and in Network Management. Today, there is a pronounced trend towards packet-based technology in telecommunications. This requires interworking between the older circuit switched and the newer packet switched networks. Java based component software can offer the platform independent support necessary.

The overriding advantage of Java is its platform independence. There is hardly any other area where this is more important than in telecommunications. There is a vast array of technologies and equipment, both wireline and wireless in existence in today's networks. Integration of technologies is a very pressing objective. The overriding objective of Java's use in Telecommunications is to provide a service environment which is independent of the underlying network technology. By separating the service logic and its access from the technology dependent networks, an open interface can be created for the development of services. For example, the underlying network may be a cellular mobile network, or a packet, or circuit based network. The aim is to provide a service environment (A Java API) which provides a standardized access to the available services. In this way the market for telecommunication services is shifted from the many proprietary systems in current use to a single, open, distributed environment, somewhat akin to today's Internet, where myriads of new application can flourish in this open environment. This also creates opportunities for independent software suppliers, service providers, protocol stack suppliers, and carriers in the provision of standardized interfaces to their products. Heterogeneity will always exist at the network level. The existing investment by network carriers and service providers must be preserved. The Java approach is to provide a standardized set of integrated APIs which provide access to the existing PSTN, ATM, IP, wireless etc networks of today. Already service provider APIs have been developed for call control, service creation and service execution environments. At the protocol level, standardized protocol interfaces are being produced to allow the interchange of applications and protocol stacks (SS7, INAP, GSM, SIP etc) [1]. Another area where Java has tremendous advantage is in Network management. The typical network today will consist of equipment from several vendors, together with proprietary O&M systems. The demands of the marketplace is for reliable, scalable, and distributed management solution which is independent of platform, supplier, operating system, and able to accommodate ever-changing network technology. Existing Network incompatible management protocols include the Common Management Information Protocol (CMIP), and SNMP, plus XML over HTTP via browsers. Again it is accepted that any solution must accommodate heterogeneity, Java has a very definite advantage in providing a common standardized interface for the Telecommunication Management Network (TMN).

Of course, Java has disadvantages. Because Java software is first compiled into intermediate architecture-independent bytecode, where at run time the bytecode is interpreted by a platform-specific Java Virtual Machine (JVM), there is naturally a performance penalty. However, with faster and faster hardware today, this penalty is far outweighed by the advantages in most application areas. In any case, this performance penalty has been addressed by the provision of the real time specification for Java, (part of the Java Community Process), which allows optimized bytecode execution, faster memory allocation and garbage collection.

Associated with Java is JINI, a specification of an architecture for the construction of distributed applications. In the context of a service environment, JINI is about how clients make contact with servers (who offer services) under dynamic network operating conditions, where servers fail, are shutdown, are overloaded etc. JINI is based on a 'discover and join' mechanism, where servers have to register their existence with a service registrar, and where they are granted a lease, which they have to renew, in order to confirm to JINI that they are still active and providing service. Clients who require service first make contact with the look up service, and from which a service API is downloaded. This enables the client to access the relevant server. There may be multiple servers providing an identical service (reliability). JINI is intended to cater for situations where services come and go, to reflect network events, it provides robustness, and adjusts to changing network conditions via leasing and re-registrations. Network Management is a distributed application. The Managment Information Base (MIB) is a set of distributed managed objects. Similarly, call control, and service execution take place in a distributed environment where clients and servers interact. The conclusion is that JINI is a distributed

Niall Gallagher and Seamus O'Shea are with the Department of Computer Science, University of Limerick, Limerick, Ireland. Phone: +353 61 21 3549 E-mails: {niall.gallagher, seamus.oshea}@ul.ie

system technology which has the capacity to confers many benefits to telecommunication applications.

In a distributed environment, servers will become overloaded from time to time. Response times will suffer, and client performance will be unsatisfactory. For example, in INAP, a Service Control Point (SCP) can be overloaded, in GSM a Home Location Register (HLR), or a Visitor Location Register (VLR) can become overloaded. Those are nodes which have real time constraints. One solution to server overloading is to provide service replication over one or more additional nodes. This replication facility should be a platform function, not a service requirement. By providing a replication service in JINI, servers that experience congestion can now spread the client request load over several nodes, thereby alleviating the congestion, and maintaining the client's quality of service.

The remainder of this paper will describe approaches to how a replication capability may be added to the JINI specification. Several options will be described.

II. Replication Models

Replication is the distribution of service objects with the goal of achieving fault tolerant highly available services. It provides a pool of objects from which to process requests. It can be categorized as either stateful or stateless. Both have distinct requirements and functions. Stateless replication provides services that are highly available without maintaining state with peer service replicas. Stateful replication requires state to be maintained between services and is thus more difficult to achieve. Stateful replication requires state to be consistent with all available replicas. This is so that when a replica fails the client can be directed to another available replica, transparently. Achieving distributed concurrency between stateful replicas requires management of resources so that objects remain strongly consistent. The solution described presents a replication system that provides transparent fault tolerance, replication management and distributed concurrency for both stateless and stateful services. The framework implements two traditional replication models active and passive.

A. Active Replication

Active replication is a replication model that involves a federation of active services. Active in the sense that each service is active and processing requests while maintaining strong consistency. With stateless objects active replication is merely a matter of communicating with one service until it fails then seamlessly redirecting to another available service. Active replication however is more difficult to achieve with stateful services. There are two primary mechanisms that can be used to achieve active replication. The first technique involves asynchronous message passing between replicas using distributed locking and leasing. Another established technique is achieved using a method called *Virtual Synchrony* [5]. *Virtual Synchrony* is where each client multiplexes requests to each active service in the federation of replicas, thus if the services are similar they should process the requests in the same manner and thus naturally maintain state. If a replica fails in the *Virtual Synchrony* technique this is not a problem. *Virtual Synchrony* employs the principal write-many-read-one. This technique is cumbersome as each service must respond to ensure that the federation maintains replica consistency. A bounded response time is needed so that if a service fails to respond it is eliminated from the federation.

B. Passive Replication

Passive replication is where the federation of replicas maintains a set of dormant services and one active primary. The primary processes every request from the consumers, however each replica can either be updated as state changes as in warm passive replication or on failure of the primary as in cold passive replication. Passive replication is convenient as it requires little management until the failure of a primary occurs.

III. Replication in Java

The Java RMI API provides techniques which enable the development of distributed objects. It provides a naming and registry service which enable objects to export there interfaces to other perhaps remote JVMs. The API provides two abstractions which enable the development of remote objects. These abstractions are the `UnicastRemoteObject` and `Activatable` objects. Both abstractions provide an `exportObject()` method which exports the object to the RMI runtime and thus makes it available for remote invocations.

Much like CORBA IDL, Java RMI provides a service contract in the shape of a remote interface. This interface enables clients to invoke methods on the remote server object. A `RemoteStub` object implements this service contract to enable method invocations to be marshaled and transported (via TCP) to the remote RMI runtime where it is unmarshaled and dispatched to the correct local object. The `RemoteStub` methods are invoked in much the same way as a local Java object. The actual mechanism of marshaling and communicating the information over TCP is handled by a `RemoteRef` and is abstracted from the consumer. There is however a certain lack of transparency. The service interface must extend the `java.rmi.Remote` interface and each method must throw a `java.rmi.RemoteException`. Java RMI parlance is thus

```
try{
   service.someMethod(argument);
}catch(RemoteException e){
   /* invocation failed */
}
```

Once the consumer of a service receives and unmarshals the `RemoteStub` it creates a TCP connection to the remote service and translates method invocations on the stub object into Java Remote Method Protocol messages (or some proprietary protocol) which marshals the method invocation and sends it and its arguments to the remote server object. The RMI runtime unmarshalls the JRMP stream and extracts an `ObjID` [8] which is used to locate the live Java object using an object table. Thus a method of communication between remote objects is achieved providing an architecture which enables objects to communicate in much the same was as local objects by simply invoking methods.

JINI is a distributed object system built on the Java RMI architecture described. It provides a framework of services which enable the construction of reliable distributed systems using leases to recover from service failures [10]. It enables discovery of services and federation of service registrars using a multicast mechanism. This enables service objects to publish service interfaces with the JINI service registrar, which provides all discovery and management of published interfaces. The RMI architecture coupled with JINI provides an attractive platform for development of a replication system. It enables the development of services that can be discovered and deployed in an *ad hoc* network environment. Although there are several implementations for fault tolerance in the CORBA community mainly through the FT-CORBA specification [6],[4],[3] these are often heavy, complex, and lack transparency. Replication with a pure Java implementation and with no modification to the existing RMI architecture or tools [7] is a more desirable technique as it offers transparency, simplicity, and portability.

A. Interception

Interception is a technique that has been used in other replication systems such as [6]. The Interception strategy in RMI involves the use of a custom socket factories [8] which can be used to intercept and modify JRMP messages in such a way that the JRMP stream can be used with an arbitrary remote server object. The custom socket factories can be given to the remote object through the `exportObject()` method of the various remote object abstractions discussed previously. This exposes an opportunity to intercept the JRMP messages from the `RemoteRef` object as they are marshalled, and modify them in such a way that they can be multiplexed or multicast to service replicas in a transparent manner.

Interception and modification of JRMP messages can be done by subclassing the `java.net.Socket` object returned from the client socket factory. The subclass must provide a decoding mechanism which can translate the JRMP messages in such a way that it can be presented seamlessly to another remote object. This is necessary as the JRMP stream encodes various information which is otherwise not made available by the RMI API. When a remote object is exported it registers a unique object identifier with an object table which is used by the RMI runtime to identify the object. This object table is consulted when a JRMP stream is opened to the remote JVM. The stream encodes an *EndpointIdentifier*[8] and a `ObjID` which are used to identify the JVM location within the Internet and the object within the JVM. A method invocation will not be accepted if it does not contain the correct *EndpointIdentifier* and `ObjID`. An interception strategy can thus be developed by publishing a service interface object with its `ObjID` and *EndpointIdentifier* so that the JRMP stream can be modified for each service object.

There is however several problems with the interception technique using custom socket factories. Problems arise when masking of failures is required, this is necessary so that the client remains unaware that there has been a failure. It is simply redirected to another replica as in passive replication or the response from an active replica that succeeded with the request is returned and the failed service is rejected from further communications. This involves the requirement of caching each method invocation that the is made so that if the network or service fails during the processing of a request the message can be resent to another replicated service. There is also a significant performance hit as the JRMP stream is marshalled and unmarshalled for each intercepted communication and also by the `RemoteRef` object. The technique is bound to a specific protocol JRMP and so does not work for `RemoteStub` objects generated to handle IIOP, SOAP, or other proprietary protocols.

B. Custom proxies

A custom proxy implementation involves the development of a proxy object that wraps several stub objects or perhaps wraps an interface to an object group where stubs are registered. The custom proxy object involves implementing an object wrapper that implements the desired interface, and so is transparent. It should capture failed RMI invocations and redirect the request to another active replica by querying the object group for an active replica. This however is an unattractive mechanism as it involves the implementation of the server, the remote service interface, and the proxy functions for redirection and *Virtual Synchrony*. Although this could be alleviated by a set of abstract base classes for the proxy object it requires much investment.

C. Reflection

The reflection mechanism combines transparency and simplicity, it involves the interception of the method invocation at the object level which avoids the problems of interception of JRMP or IIOP protocols. The Java Reflection mechanism `java.lang.reflect` provides a unique way to develop RMI proxies. These proxies remain as transparent as any conventional RMI proxy (in fact even more so) and also mask server failures to provide a transparent simple replication proxy mechanism. Java introduced the `java.lang.reflect.Proxy` object and the `java.lang.reflect.InvocationHandler` in the 1.3 release of the Standard Development Kit. These provide a facility that enables a handler to receive a callback from an interface implementation. This is not unusual for an object oriented language. However, this interface implementation is generated at runtime. So with an interface and no corresponding implementation an instance of that interface can be generated that delegates the processing of the invoked method to the `InvocationHandler`. This allows an extremely dynamic mechanism for developing transparent method handlers.

This mechanism as it turns out pairs quite well with the RMI stub mechanism. This enables the `InvocationHandler` to receive callbacks from the interface instance with a description of the method invoked and also the arguments to that method. Using this mechanism, the `InvocationHandler` now becomes the ideal location to introduce the functions required for transparent replication of remote objects, with undetectable failures. Translating the the local object invocations to RMI invocations is simply a matter of generating a method hash [8] and invoking the `RemoteRef` with a selected `RemoteStub` object. This enables both IIOP and JRMP protocols to be used as well as any proprietary protocols. A replication framework can now implement a set of `InvocationHandler` objects which can be used to implement the functions required for passive and active replication. These handlers can be created using an object registry so that invocations are directed to objects that are members of a particular federation.

Distributed objects can now be developed using the traditional RMI architecture with no modification or extension to the framework or to the tools. Both `Activatable` and

`UnicastRemoteObject` implementations can be developed as usual. All that is needed is an object registry which can be used by the `InvocationHandler` to acquire live replicas. This technique can actually be used to replace remote service objects that have been deployed without fault tolerant replication, at runtime, unknown to the remote service object or the consumer of that service. The consumer of the service only needs to be given an interface object that is generated by the `java.lang.reflect.Proxy` mechanism, and method invocations can proceed as usual. The JINI service registrar provides an opportunity for these service interface objects to be published so that the consumers of services can be assured a fault tolerant highly available service in an *ad hoc* network environment.

IV. Replication Management

It has been clearly observed that Java has a unique ability to implement transparent proxy objects that mask failures in an RMI environment. It is also clear that Java presents itself as a uniquely capable platform from which to build fault tolerant highly available replication services using the RMI architecture and JINI. What is needed however is a replication management system that enables the transparent replication of services across the system. This is needed so that the services can dynamically deploy themselves across the system without the need for intervention. This enables the system to dynamically configure itself to manage load and recover from failures. A replication management system needs to be able to retrieve service replicas dynamically without ever having known about them, transfer state from live replicas, and begin processing requests.

There is also a need to enable such a system to detect failures and overloading. This will provide a stimulus for the system to replicate services. In general it is desirable to have a system that does not depend on hierarchies and that can be deployed in a JINI *ad hoc* network environment, This enables services to manage their own replication by posting requests to replication management service allowing the system to balance and control movement of services. The framework described provides a set of replication management services that control the replication of services in times of overloading and when failures occur.

A. Open Framework

The replication services described in this section define a set of abstractions that can be extended by the developer to produce fault tolerant services. The framework defines a set of features that enable both passive and active replication models to be applied to distributed objects. Traditional RMI distributed objects can be given fault tolerant semantics without any modifications. The replication model takes advantage of the `JavaSpace` service provided by JINI and achieves transparent fault tolerance with the use of `java.lang.reflect.InvocationHandler` objects.

A.1 Transparent Proxies

Developing transparent proxy objects which can be used to publish service interfaces is achieved by providing a set of `InvocationHandler` objects. The framework includes a `PassiveHandler` and an `ActiveHandler`. These both provide functions that enable the client to invoke methods of the service interface proxy transparently. The handlers use a `Federation` to group replicas with the same identity. So when a client invokes the service proxy the method invocation is delegated to the `InvocationHandler` objects. Depending on the replication model used, the `Federation` is used to retrieve a set of *active* replicas. The handler can now either iteratively invoke the methods of the `RemoteRef` objects for active replication or choose one and invoke its methods for passive replication. The client thinks that it is communicating on a one-to-one basis with a traditional RMI distributed object, it is completly transparent. The service interface proxies are created using the `java.lang.reflect.Proxy` object. This generates a implementation of the service interface that delegates the client invocations of the service interface to the `InvocationHandler`. This now enables failures to be masked so that redundant replica objects can register themselves with a `Federation` and become a part of the replication system.

Federating the replica services provides an opportunity for load balance amongst the replicas. The use of the invocation handler objects means that invocation of a particular service is dependent on the functions that the handler provides. This enables the `InvocationHandler` to measure the invocation times of particular methods and provide feedback to the `Federation` service. Distributed concurrency later mentions how replicas that maintain state using an active replication model can use one-to-one method invocations to achieve parallelism between the replicated services while still maintaining state with peer replicas. Using the notion of service monitoring based on response times provides the federation with a unique ability to configure service interface proxies that use the `InvocationHandler` to direct requests at the quickest responding services based on the feedback of the handlers. This does not restrict the services to achieve balance based on response times, as network latency can play a large role in the response times of a service. It is left to the service developer to subclass the `InvocationHandler` objects provided to include other measurement functions if required. The implementation of the handlers is such that the developer is unaware of the balancing functions unless a specialization is needed.

A.2 Replication Servers

The traditional RMI distributed objects are `Serializable`, this means that their state can be persisted. However a remote object can be persisted only before it is exported to the RMI runtime by explicit use of the `exportObject()` method or by subclassing either of the abstractions provided by the API. In order to achieve replication the service must be able to persist state. This enables the service to be transported to a remote location. The serialization of the remote object will transfer the state of the live object as it is. This will enable stateful objects to be replicated (or at least aid the process). This is fine for simple service objects, however if the service needs strong consistency a set of interfaces needs to be implemented to simplify the process and provide the developer with more control over the service implementation. Serialization will enable the state of the object to be transferred but non-volatile data also needs to remain consistent. Replication servers are provided to enable the objects to be transferred to remote locations. They export a service interface that allows the service state to be transferred.

The replication server is a `Remote` object that resides on a node within a connected network. This server makes use of the JINI framework to publish its service interface. The JINI environment also provides a services which enable a balanced distribution of replicas within the network, the `JavaSpace` [2]. The purpose of the replication server is to download, initialize, and start replica services. It publishes a service interface that enables the transfer of both functionality and state of arbitrary service objects. This is achieved using a `Driver` object which abstracts the technique used to transfer the service to the replication server, so serialization using the Java Serialization API is optional. Once the service has been transferred state is synchronized using an `init` method which takes a `Context`. The `Context` provides the replica with a view of the host and enables the replica to transfer non-volatile state from a live replica.

Balance is an important feature in this system. If a service requires replication it could contact the JINI service registrar and download a suitable interface for the replication server. However this would impose an unfair balance of load on a particular replication server and as is discussed later reduces the ability of replica movement. A technique which balances and coordinates replicas involves posting the request for replication using a `JavaSpace`. The service acquires the space from the JINI service registrar and posts its request for replication along with some meta data which describes the location it would like to be replicated and the identity of the replica. Each replication server has access to the `JavaSpace` and reads the posted requests. If a replication server decides that it is suitable based on perhaps security policies or location it will `take()` the request from the `JavaSpace` and initiate transfer of service and state. This enables replicas to move towards areas of high request. If replication servers are deployed throughout a network moving service replicas towards a node that is close to areas of high request is only a matter of maintaining a journal that documents the origins of requests. When it wants to replicate it includes an origin of high request.

Removing the `JavaSpace` as a single point of failure is simply a matter of implementing a passively replicated transient `JavaSpace` such as the `TransientSpace` from the `com.sun.jini.outrigger` package and using small lease times which can detect failures quickly. Deploying a set of dormant spaces throughout the system is only a matter of providing a `Driver` implementation that wraps a method of starting a `JavaSpace`. Any service can use this implementation. The service interface for the replication manager is

```
interface Replicator extends Remote {

   public Object replicate(Driver driver)
      throws RemoteException;
}
```

This service interface provides a method that enables a `Driver` object to be used by the replication manager to download the service and initiate it. The method returns an `Object` which is the service interface of the newly replicated service. To replicate a required service such as a space the service contacts a replication manager and invokes the `replicate` method using the `Driver` implementation for the transient space. Once the replication is complete it casts the returned object to the service interface and uses the replicated service as desired. Each service must implement a service interface in order to be replicated, this interface is

```
interface Replica extends Serializable{

   public void init(Context context)
      throws RemoteException;

   public void start()
      throws RemoteException;

   public void stop()
      throws RemoteException;
}
```

Once the replica has been transferred the replication manager invokes the `init`, `start`, and `stop` methods in sequence. For a passive model the `start` method may not be invoked until it is elected the primary. The replica is terminated and removed from the `Federation` when the `stop` method is invoked.

A.3 Replication

The replication of a service is achieved using the service interface described above. However there are two proxy interfaces needed, one that interfaces directly with the replication server and one that can post requests into the `JavaSpace`. The use of a second proxy also enables security to be used in the system, it abstracts the details of contacting the service registrar and provides a simpler interface to the system. The replication of a service can thus be achieved by the following sequence of events

Contact JINI registrar Once the service notices that it is overloaded, or when the `Federation` notices that there are replica failures, services need to be replicated. To achieve this the JINI service registrar is contacted and a proxy for the replicated `JavaSpace` is acquired.

Post request The `Replicator` proxy uses the `JavaSpace` to post the request for replication. Each of the replication managers has access to the space and receives notification of the new posting. If the replication manager feels that the request is suitable it handles the request.

Handle request A replication manager will take the entry from the `JavaSpace`. The request in thus removed completely from the `JavaSpace`. This entry object is then used to contact the service to initiate replication.

Service download The Java Serialization API enables the Java objects to be streamed over an `OutputStream` to the replication manager. The replication manager will however use a `Driver` object which manages the downloading of the service.

Initialization Once the service object has been downloaded by the replication server it is given a `Context` object using the `init` method which enables it to copy state from another live replica. This however may not be needed in the case of stateless objects or if the `Driver` object is implemented so as to download all the live service data.

Execution Once the state of the service has been assured the service is started, or in the case of passive replication is dormant until elected as the primary.

A.4 Distributed Concurrency

The previous description of a replication management system described a dynamic federation of replication servers that are used to enable replicas to be transported in a dynamic manner across the system. However as well as fault tolerance requirements there are also performance requirements. The described system is capable of distributed concurrency in many areas. The use of the `java.lang.reflect.InvocationHandler` provides a mechanism for increasing the security and concurrency of the fault tolerance federation.

The use of security and distributed concurrency seem distinct but combined can produce seamless parallelism between peer service replicas. The use of the *Virtual Synchrony* model of maintaining state between replicas is a slow and cumbersome one. However a method of concurrency can be achieved by the use of the write-many-read-one *Virtual Synchrony* model and the write-one-read-one passive replication model. The use of the `java.lang.reflect` facilities enables the dynamic configuration of proxy objects. Proxy objects can be transported with *Virtual Synchrony* functionality or in a manner that simply redirects to another replica to mask failures. These can be used in a manner that involves authentication. Only certain principals in the system can issue state changing invocations. So when a consumer of a particular service requests a service proxy it must precede authorization. A one-to-one proxy is given to principals with read only permission and a *Virtual Synchrony* proxy is given to principals with write permission. Thus state is maintained and distributed concurrency is also achieved in a fault tolerant way to provide a highly available high performance federation of replicas.

V. Conclusion

A useful replication framework should provide features that do not rely on hierarchies and that perform transparent replica management. It should be simple and should require little intervention from the service developer. However it should enable the developer of the services to exercise as much control over the federation as required. It should contain a set of simple core features that are required for replication and a set of convenience features that should be separate from the core facilities to reduce the complexity of the framework. It should also feature a set of interfaces which must be implemented to provide strong consistency, replica deployment, and expose the developer to the semantics of the replication system. This enables the developer to process requests in a manner that is consistent with peer replicas so that for example transaction x executed with replica a produces the same results as transaction x executed with replica b. The `java.lang.reflect` package provides unique tools that enable RMI to be used for the development of distributed replication without any alteration to the RMI protocols or to the RMI architecture, coupled with JINI the services can be deployed in an *ad hoc* network environment.

References

[1] Thomas C. Jepsen, Farooq Anjum, Ravi Raj Bhat, and Douglas Tait, *Java^{TM} in Telecommunications: Solutions for Next Generation Networks*, John Wiley and Sons, 2001.

[2] Eric Freeman, Susanne Hupfer, Ken Arnold, *JavaSpaces^{TM} Principles, Patterns, and Practice* Addison-Wesley, 1999.

[3] Carla Marchetti, Massimo Mecella, Antonio Virgillito, and Roberto Baldoni, *An Interoperable Replication Logic for CORBA Systems*, DOA 2000.

[4] Balachandran Natarajan Aniruddha Gokhale, Shalini Yajnikm, and Dogulas C. Schmidt, *DOORS: Twards High-Performance Fault Tolerant CORBA*, DOA 2000.

[5] Silvano Maffeis and Douglas C. Schmidt, Constructing Reliable Distributed Communications Systems with CORBA, IEEE Communications Magazine, 1997.

[6] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, V. Kalogeraki, and L. Tewksbury, *The Eternal System*, Department of Electrical and Computer Engineering, University of California.

[7] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik, *Filterfresh: Hot Replication of Java RMI Server Objects*, COOTS 1998.

[8] Sun Microsystems Inc, *Java^{TM} Remote Method Invocation Specification*, 2002.

[9] Sun Microsystems Inc, *JavaSpace^{TM} Service Specification*, 2001.

[10] Sun Microsystems Inc, *JINI^{TM} Technology Core Platform Specification*, 2001.